UNITED STATES PATENT APPLICATION FOR

SYSTEM AND METHOD FOR RECURSIVE PATH ANALYSIS OF DBMS PROCEDURES

Inventors:

John K. Vincent

Igor (nmi) Cherny

Prepared by:
Erwin J. Basinski

Morrison & Foerster
425 Market Street
San Francisco, California 94105-2482

SYSTEM AND METHOD FOR RECURSIVE PATH ANALYSIS OF DBMS PROCEDURES

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is related to application serial number _____ entitled "**METHOD AND APPARATUS FOR EXECUTING STORED CODE OBJECTS IN A DATABASE**," filed December 22, 1998.

TECHNICAL FIELD

The invention relates to the field of computer-related systems and methods. More specifically, the invention is a system and method for automatically generating complete dependency information of DBMS Stored code objects.

BACKGROUND ART

It is desirable to provide efficient and cost effective processes for generating queries to a data base management system. In today's world of Relational Data Base Management Systems (RDBMS), data bases can store objects and such objects can invoke other objects. Processes for invoking such objects involve invoking procedures which may themselves invoke other procedures to several levels. This nested complexity makes the debugging and testing of such objects very time consuming and machine inefficient.

In the past, many attempts have been made to automate the process of generating the complete dependencies of programs and to use it for compiling and debugging code. One such attempt, involved binding together pre-compiled subroutines to form a complete host procedure object code, as described in US Patent No. 4,330,822 titled "Recursive System and Method for Binding Compiled Routines". Another such attempt, involved reducing the compilation time of modules using one level of module dependencies, as described in US Patent No. 5,586,328 titled "Module Dependency based Incremental Compiler and Method."

Yet another attempt, involved generating complementary source code to resolve external dependencies of program units to facilitate unit testing, as described in US Patent No. 6,651,111 titled "Method and Apparatus for producing a software test system using complementary code to resolve external dependencies". Yet another attempt, involved building an object oriented software program using compiler generated direct dependency information described in US Patent No. 5,758,160 titled "Method and apparatus for building a software program using dependencies derived from software component interfaces". In all of these cases only the direct dependencies of the modules concerned are used. In a database programming environment, spending an inordinate amount of time finding the dependencies of program units using a compiler is undesirable, since that information is directly available from the database catalog. Many other US patents describe various debugging and testing systems but none of these which is known to Applicant provides the method and system of the present invention for automatically generating the complete dependencies necessary to debug code objects.

It would be advantageous to have a method for automatically generating debug versions of a subprogram and all its dependencies. The method should allow fixing coding errors much faster by eliminating the need for generating debug versions of all dependent subprograms in a manual fashion. The method should also allow detecting potential runtime errors, before the subprogram is debugged or executed. This would allow the elimination of some of the run time errors that can be very hard to detect in a production environment. The method should also allow programmers to visualize a graphic representation of the complete dependencies of subprograms. The method should also allow programmers to visualize INVALID database objects in the dependency graph. The method should also allow visually identifying cyclic dependencies. This would eliminate the need for programmers spending time figuring out the actual dependencies, the nature of such dependencies and the validity of such dependent objects. This process involves a combination of browsing the source code and looking up the database catalog for specific dependency information. This manual

process can be exhaustive, since multiple levels of dependencies are prevalent in database development environments.

For example, the technical problem can be appreciated with the following additional information:

PL/SQL is a complex language, that allows complex inter-dependent code to be developed. Code modules in PL/SQL can reside in separate library units that reference one another. There are four main types of library units:

1. Packages are true libraries that can include one or more stored procedures and one or more functions. Each of those objects can be public or private. Public objects (exposed in the package spec) can be invoked from other library units. Private objects can be invoked internally within the package. Each object (procedure or function) can invoke other objects within the package or external to it.

2. Stored procedures are stored objects that include only one callable module. Code in a procedure can invoke external objects.

3. Functions are stored objects that include only one Callable module. Code in a function can invoke external objects. Functions differ from procedures by returning a value.

4. Triggers are stored objects that are fired by the database engine when various events occur. Triggers can invoke external objects.

Oracle8™ also has Types which are abstract data types that can hold data members and member functions.

The debug process involves an object and all its dependencies. If a logical problem exists with a value returned or set by a called object, then the coding error might exist in either the called object itself, or one of the called objects. A true debugger should let the developer step through the code traversing dependencies at will, without any special effort. The level of complexity in large applications can easily reach 5 to 10 levels of dependency and the dependency tree can include hundreds of objects.

The alternative for the debugger automatically detecting all the dependencies is for the user to manually analyze the dependencies for the objects and then perform a process that alters all those objects to debug mode, so that they can be debugged. In the example mentioned above, with hundreds of dependent objects this process is tedious and time-consuming.

Alternatively, users can compile all their objects in debug mode, but this again is not optimal, since upon completion of the debug phase they will have to re-compile everything again (for production). Then, for every bug discovered later, the same process is required.

Some RDBMS try to provide assistance in handling these problems. For example, Oracle provides the 'connect by' clause to generate an implicit tree as the result of SQL query. This method of querying could be applied to provide a partial solution to the problem of generating the complete dependency tree of a stored code object. The method can only provide a partial solution because of the way some of the Oracle code objects behave. Specifically, packages are implemented in Oracle as ***two distinct code objects*** - a package specification and a package body. Applying the 'connect by' clause above will result in a tree that will contain dependencies of all the package specifications in the dependency tree but not of the corresponding package bodies. A variation of the 'connect by' clause that also tracks package body dependencies cannot be constructed because SQL does not provide a way of saying not to connect any further, if a condition is satisfied - i.e. to prevent infinite recursion in the dependency tree, it is imperative to support a way by which, if a dependency already occurs in the tree in the parent path, we should not proceed to get the dependency of the object again.

The present invention is an efficient and effective solution to the technical problem of retrieving all object dependencies from obects stored in a RDBMS. The solution to this technical problem developed by applicants uses a query that is called recursively. An array is used to track the parents so that the graph can be reconstructed. At each step, it is determined whether the dependency already occurs in the graph. If it occurs, the recursion is stopped.

DISCLOSURE OF THE INVENTION

A system, method and database development tool are disclosed for automatically generating the complete dependencies of a stored code object in a database by applying a set of recursive procedures and parsing the source code of the code objects.

Also a method for generating a cyclic graph of dependencies based on the complete dependency information and their relationship with one another is claimed. Additionally claimed are a method for generating debug versions of stored code objects and all its dependencies. Also claimed is a method for identifying potential run-time errors based on the information about the validity of the dependent code objects. Also claimed is a method for identifying the cyclic dependencies of a database code object. Also claimed is a method of debugging code objects in a database using the complete dependency graph of the particular code object. Also claimed is a method of developing database programs comprising a computer system and a program code mechanism for automatically generating complete dependencies of stored code objects.

Other embodiments of the present invention will become readily apparent to those skilled in these arts from the following detailed description, wherein is shown and described only the embodiments of the invention by way of illustration of the best mode known at this time for carrying out the invention. The invention is capable of other and different embodiments some of which may be described for illustrative purposes, and several of the details are capable of modification in various obvious respects, all without departing from the spirit and scope of the present invention.

BRIEF DESCRIPTION OF THE DRAWINGS

The features and advantages of the system and method of the present invention will be apparent from the following description in which:

Figure 1 illustrates a flowchart of the overall flow of a preferred embodiment of the present invention.

Figure 2 illustrates a flowchart showing the general processing flow of the recursive analysis steps of the preferred embodiment.

Figure 3 illustrates a flowchart showing the steps used to incorporate dependencies of package bodies into dependency graph.

Figure 4 and 5 together illustrates a flowchart showing the steps used to incorporate dependencies on database triggers and their dependencies into the dependency graph.

Figure 6 illustrates a flowchart showing the steps involved in getting the dependencies of a code object that does not take into consideration the indirect dependencies such as the dependencies of package bodies and dependencies on triggers.

Figure 7 shows a sample dependency-tracking array, which is the base data structure used for storing the data elements of the dependency graph.

Figure 8 is a sample object tree used for purposes of illustration.

Figure 9 illustrates a representative general purpose computer configuration useable as either a client PC and/or a server in the preferred embodiment of the present invention.

BEST MODE FOR CARRYING OUT THE INVENTION

The present invention provides a method and apparatus for generating the complete dependency graph of a database code object. The ability to debug a code object is a fundamental requirement for a developer. This allows verifying the logic of code objects as they are being developed.

In the following description for purposes of explanation, specific data and configurations are set forth in order to provide a thorough understanding of the present invention. In the presently preferred embodiment the invention is described in terms of an Oracle Integrated development Environment (IDE). However, it will be apparent to one skilled in these arts that the present invention may be practiced without the specific details, in various Data Base systems such as Sybase, Microsoft SQLServer, UDB, DB2, Informix, etc. In other instances, well-known systems and protocols are shown and described in diagrammatical or block diagram form in order not to obscure the present invention unnecessarily.

ADDITIONAL BACKGROUND INFORMATION

As the data stored in DBMS grows more complex, the SQL queries used to retrieve and/or update the data are growing more complex as well. This has lead to the addition of procedural and object oriented extensions to the standard SQL language. PL/SQL™ is a transaction processing language that offers procedural and object oriented extensions to program against the Oracle database. Other databases provide similar procedural and object oriented extensions. The increasing complexity of these languages has lead to the development of special debuggers that run the SQL language statements under tight control and allow the developer to interact with and analyze the statements as they are executed.

The combination of interactive debugging with code objects that reside on the server has lead to a problem where the debugger must be configured for the procedure that is being debugged and all its dependent procedures. Specifically, the 'step into' operation that allows you to step into dependent objects will work only if those code objects are configured to be debugged. So the user either has to identify and configure all procedures that will be used, including procedures called by those procedures or else they give up the full debugging control.

This is a cumbersome process and frequently leads to frustration on the part of the developer.

PL/SQL is a complex language that allows complex inter-dependent code to be developed. Code modules in PL/SQL can reside in separate code objects

that reference one another. Code objects in PL/SQL take the form of one of the following:

a) A PL/SQL stored procedure, which is a routine that can take input parameters and return output parameters,

b) A PL/SQL stored function, which is a routine that can take input parameters and return output parameters. In addition stored functions can be used in an expression and returns a value of a particular type declared in the specification,

c) A PL/SQL stored package specification, which forms the specification for a collection of stored functions and procedures identified as a single named entity,

d) A PL/SQL stored package body, which forms the implementation for a collection of stored functions and procedures identified as a single entity,

e) A PL/SQL stored type specification(Oracle8™ only), which forms the specification for a collection of data elements and member functions identified as a single named entity,

f) A PL/SQL stored type body(Oracle8™ only), which forms the implementation for a collection of member functions identified as a single entity,

g) A PL/SQL stored trigger, which is a routine that gets executed automatically, when a data manipulation statement is executed in the database server and

h) An anonymous PL/SQL block, which is an unnamed set of PL/SQL statements that can be executed on the server.

Databases support object-oriented programming in different ways. Oracle7™ supports this programming methodology through PL/SQL stored Package Specifications and PL/SQL stored Package Bodies. Although the syntax of such support is different on different databases, the fundamental concept of having a specification which can represent an abstract data type and an implementation which implements the specification is supported in all databases.

(Oracle, Oracle7, Oracle8, and PL/SQL are trademarks of Oracle Corporation).

The debug process involves an object and all its dependencies. If a logical problem exists with a value returned or set by a called object, then the coding error might exist in either the called object, or one of the dependencies of the called objects. A true debugger should let the developer step through the code traversing dependencies at will, without any special effort. The level of complexity in large applications can easily reach 5 to 10 levels of dependency and the dependency tree can include hundreds of objects.

The alternative for the debugger automatically detecting all the dependencies is for the user to manually analyze the dependencies for the objects and then perform a process that alters all those objects to debug mode, so that they can be debugged. In the example mentioned above, with hundreds of dependent objects this process is tedious and time-consuming.

Alternatively, users can compile all their objects in debug mode, but this again is not optimal, since upon completion of the debug phase they will have to re-compile everything again (for production). Then, for every bug discovered later, the same process is required.

OPERATING ENVIRONMENT

The present invention operates in and as a part of a general purpose computer unit which may include generally some or all of the elements shown in **Figure 9**, wherein the general purpose system **201** includes a motherboard **203** having thereon an input/output ("I/O") section **205**, one or more central processing units ("CPU") **207**, and a memory section **209** which may have a flash memory card **211** related to it. The I/O section **205** is connected to a keyboard **226**, other similar general purpose computer units **225, 215**, a disk storage unit **223** and a CD-ROM drive unit **217**. The CD-ROM drive unit **217** can read a CD-ROM medium **219** which typically contains programs **221** and other data. Logic circuits or other components of these programmed computers will perform series of specifically identified operations dictated by computer programs as described more fully below.

THE PREFERRED EMBODIMENT

The present invention provides a method and apparatus for generating the complete dependency graph of a stored code object. In a preferred embodiment, the method generates the complete dependency graph of a stored code object in an Oracle database. The method takes into consideration, getting the dependencies of object oriented code objects that have implementations that are separate from specifications. In the preferred embodiment, this involves incorporating the dependencies of Package and Type Bodies in addition to the dependencies of Package and Type Specifications. The method also takes into considerations dependencies on database Triggers. Triggers are code objects that are executed automatically as a result of executing a Data Manipulation Statement (DML) that modifies data in a particular table. Other code objects are not directly dependent on triggers. However, an indirect dependency is possible, when the code object executes a DML statement that automatically executes ("fires") the trigger. The present invention accomplishes this through a process described by the following algorithm:

1. Using a recursive procedure that returns dependencies of procedural code objects and dependencies of specifications of object oriented code objects, based on a query against the database that returns a single level of dependencies.

2. Using the recursive procedure in Step 1 to determine the dependencies of implementations of object oriented code objects, based on a query against the database that returns a single level of dependencies.

3. Using a source code parsing procedure to identify DML statements that can "fire" triggers. Specifically, the parser looks for UPDATE, DELETE and INSERT statements. The parser identifies the type of the statement and also the database table. The algorithm then checks whether a corresponding trigger exists in the database. This identifies the dependencies on triggers of any stored code object. The flowchart in Figures 4 and 5 explains a variation

of this algorithm that starts with the dependencies on tables working backward to find the dependencies on triggers.

4. Using the recursive procedure in Step 1 to determine the dependencies of triggers found in Step 3, based on a query against the database that returns a single level of dependencies.

5. Repeating Steps 2 through 4 until there are no new dependencies.

The general flow of the preferred embodiment is now described with reference to the Figure 1.

The user interface element component 11 is used to select a code object from the database. The selected code object provides the information required in generating the dependency information. Component 12 is used to generate the complete dependency graph of the selected object. It generates a data structure that holds the complete dependency information hereby termed a tracking array. The tracking array is used by Component 13 to generate a User Interface element that shows the complete dependency graph. Component 14 scans the tracking array and compiles the objects in debug mode. This step ensures transparent "Step Into" operation from the Debugging facility. If any of the objects that are compiled in debug mode are INVALID, Component 15 shows the complete dependency graph with INVALID objects highlighted. INVALID objects in the complete dependency graph identifies code paths that can result in runtime errors. This functionality is indispensable for database programmers since otherwise a tremendous amount of time may be spent trying to identify the source of a run-time error. Component 14 scans the dependency array and for each object in the array, checks whether that object is part of a cyclic dependency. A well-known graph traversal algorithm is used for the purpose. Cyclic elements of the graph are then highlighted in a User Interface component that displays the dependency graph. Component 16 uses the tracking array generated in Component 12 and the objects compiled in debug mode to launch the Debugger.

The general technical solution to the problem is now described with reference to Figure 2 through 5.

Component 21 in Figure 2 applies the flowchart in Figure 6 to generate a tracking array of dependent code objects. The array data structure shown in **Fig.7** contains all the information necessary to draw a dependency graph. The algorithm does not take into consideration dependencies on triggers as well as dependencies on implementations of object oriented code objects. .As a result, after the initial array data is generated in block 21, flow then proceeds to component A 23 which is described below with respect to Figure 3 wherein the dependencies on implementations of object oriented code objects are identified. Then the system flow proceeds to component B 24 which is described in below with respect to Figure 4 wherein any dependencies on triggers are evaluated. After determining that no new nodes were found (component 22) the routine is completed with the array containing all dependencies of the target object.

Referring now to Figure 3, Component 31 selects the next code object starting with the first one in the tracking array. Component 32 checks whether the code object is a Package or a Type. (Packages are defined and described in Chapter 7, Oracle7™ Server Application developers Guide, which is hereby incorporated fully herein by reference. Types are defined and described in Oracle8 SQL Reference, Release 8.0, which is hereby incorporated fully herein by reference). If it is a package or type, Component 33 applies the flowchart in Figure 6 again to incorporate the dependencies of the package body or the type body respectively into the tracking array. Component 34 then checks whether this is the last object in the tracking array.

Component 41 in Figure 4 selects the next code object starting with the first one in the tracking array. Component 42 gets the dependencies on database tables of the code object using a database query against the catalog. Component 43 selects the next database table dependency starting with the first one. Component 44 gets the dependencies on database triggers of the database table. (Triggers are defined and described in Chapter 8, Oracle7™ Server Application developers Guide, which is hereby incorporated fully herein by reference). Component 45 selects the next database trigger dependency starting with the first one. Component 46 parses the source code of the code object to verify whether a

Data Manipulation Statement (DML) that "fires" the trigger is present in the source code. If such a statement is present, component 52 in Figure 5 applies the flowchart in Figure 6 again to incorporate the dependencies of the trigger into the tracking array. Component 53 in Figure 5 checks whether this is the last trigger dependency. If No, Control returns to Component 45 in Figure 4 that selects the next trigger dependency. If Yes, Component 54 in Figure 5 checks whether this is the last table dependency. If No, Control returns to Component 43 in Figure 4 that selects the next database table dependency. If Yes, Control shifts to Component 22 in Figure 2 that checks whether a new node was added during application of the algorithms in Figures 3 and 4. If No, the algorithm terminates and the tracking array contains the complete dependency information of the selected code object.

A more detailed description of the basic recursive process is now described with reference to Figure 6 and Figure 7.

The current code object is passed in as an argument to the algorithm in Figure 6. Component 61 in Figure 6 adds the current code object as a new item in the tracking array. A sample item in a tracking array is depicted in Figure 7. As illustrated, the tracking array consists of the following elements – the object id number 71 that uniquely identifies the object in the database, the object name 72, the owner of the object 73, a flag indicating whether the object is residing in the current database 74, the type of the object 75 and a variable number of child object ids 76. The child object ids 76 track the direct dependencies of a particular code object. Referring again to Figure 6, Component 62 queries the database catalog to get the direct dependencies of the current object. Component 63 then gets the next direct dependency starting with the first one. Component 64 checks whether a dependency was found in Component 63. If none was found, the algorithm terminates. If one was found, Component 65 then updates the tracking array element of the current object to contain this dependency. This is done by adding the object id of the dependent object as a child object id in to the tracking array element of the current object. Component 66 then checks whether this dependent object is already present in the tracking array. If Yes, Control shifts to

component 63 which gets the next direct dependency. If No, the algorithm is recursively called by passing in this dependent object as the argument.

An example, which depicts a typical object and its dependency analysis, is now described with reference to Figure 8.

Following is an example of a typical stored function:

```
1. create or replace function CheckGender return number
2. as
3. begin
4. proc1;
5. proc2;
6. proc3;
7. end;
8. /
```

The function is depicted as the node with 'id=1' in Figure 8. The function has direct dependencies on 3 procedures 'proc1', 'proc2' and 'proc3' depicted in Figure 8 as the nodes with 'id=2', 'id=3' and 'id=4' respectively. The procedure 'proc2' does not have any dependencies. The procedure 'proc3' has a dependency on a procedure 'proc6' depicted in Figure 8 as the node with 'id=10'. For illustrative purposes, the source code of 'proc1' is shown below:

```
1. create or replace procedure proc1 as
2. begin
3. proc4;
4. insert into table1 values (1);
5. proc5;
6. end;
7. /
```

The procedure 'proc1' has direct dependencies on procedure 'proc4' and 'proc5' depicted in Figure 8 as nodes with 'id=5' and 'id=7'. The procedure also has a Data Manipulation Statement (DML) that fires an insert trigger 'trig1' depicted as the node with 'id=6' in Figure 8. The trigger 'trig1' in turn has

dependencies on procedures 'proc3' and 'proc7' depicted in Figure 8 as the nodes with 'id=4' and 'id=8' respectively. The trigger 'trig1' also has a dependency on a stored package 'pack1' depicted in Figure 8 as the node with 'id=9'. The specification of 'pack1' does not have any dependencies. However, the body of 'pack1' has a dependency on procedure 'proc1' depicted in Figure 8 as the node with 'id=2'.

In this example, the algorithm works as follows: Applying the algorithm depicted in Figure 6 (the first step in the algorithm in Figure 2) yields the basic dependency graph consisting of objects with ids 1, 2, 3, 4, 5, 7 and 10. The tracking array now has 7 items. Item 1 has object id 1 and has entries 2, 3 and 4 as child object ids. Item 2 has object id 2 and has entries 5 and 7 as child object ids. Item 3 has object id 3 and has no child objects. Item 4 has object id 4 and has entry 10 as the only child object id. Item 5 has object id 5 and has no child objects. Item 6 has object id 7 and has no child object ids. Item 7 has object id 10 and has no child object ids. Notice that the dependency of 'proc1' (id=2) on the trigger 'trig1' (id=6) is not incorporated yet. The dependencies of the trigger 'trig1' are also not incorporated yet. Applying the algorithm in Figure 3 then yields the exact same graph with no changes. Applying the algorithm in Figure 4 and 5 then yields a dependency graph that includes all the nodes in the complete dependency graph. Item 2 now has id 6 as an additional child object. The tracking array now has 10 items. Item 8 has object id 6 and has entries 8, 9 and 4 as child object ids. Item 9 has object id 8 and has no child object ids. Item 10 has object id 9 and has no child object ids. However, the dependency of the body of package 'pack1' (id=9) on the procedure 'proc1' (id=2) is not yet incorporated in the dependency graph. We then apply the algorithms in Figures 3 through 5 again, since the condition in component 22 in Figure 2 evaluates to 'Yes' – Nodes with ids 6, 8 and 9 were added when we applied the algorithms in Figures 3 through 5. Applying algorithm in Figure 3 yields the complete dependency graph – the dependency of the package body of 'pack1' (id=9) on the procedure 'proc1' (id=2) is added during this step. Item 10 in the tracking array now has a child object id 2. Applying the algorithm in Figure 4 and 5 does not change the

dependency graph. The condition in component 22 in Figure 2 now evaluates to 'No', since no new nodes were added. The algorithm in Figure 2 terminates. The tracking array now holds the information needed to generate a complete dependency graph. The tracking array is then scanned to generate the debug versions of all objects in the dependency graph. If any of the objects so compiled is INVALID, such objects are highlighted in the generated dependency graph. For each object in the tracking array a well-known graph traversal algorithm is applied to identify whether they are part of a cyclic dependency. Such cyclic paths in the graph are then highlighted. In the present example object ids 2, 6 and 9 form a cyclic path and so those nodes and the paths are highlighted. This identifies a potential infinite loop at runtime. The graph identifies the possibility that, when the trigger 'trig1' (id=6) fires and the package body code gets executed, the package body of 'pack1' (id=9) may in turn call the procedure 'proc1' (id=2) thereby resulting in a potential infinite loop.

Having described the invention in terms of a preferred embodiment, it will be recognized by those skilled in the art that various types of general purpose computer hardware may be substituted for the configuration described above to achieve an equivalent result. Similarly, it will be appreciated that arithmetic logic circuits are configured to perform each required means in the claims for processing internet security protocols and tunneling protocols; for permitting the master unit to adaptively distribute processing assignments for incoming messages and for permitting cluster members to recognize which messages are theirs to process; and for recognizing messages from other members in the cluster. It will be apparent to those skilled in the art that modifications and variations of the preferred embodiment are possible, which fall within the true spirit and scope of the invention as measured by the following claims.